

## 55. IWK

Internationales Wissenschaftliches Kolloquium  
International Scientific Colloquium



13 - 17 September 2010

# Crossing Borders within the **ABC**

**A**utomation,

**B**iomedical Engineering and

**C**omputer Science



Faculty of  
Computer Science and Automation

[www.tu-ilmenau.de](http://www.tu-ilmenau.de)

*th*  
TECHNISCHE UNIVERSITÄT  
ILMENAU

Home / Index:

<http://www.db-thueringen.de/servlets/DocumentServlet?id=16739>

## **Impressum Published by**

Publisher: Rector of the Ilmenau University of Technology  
Univ.-Prof. Dr. rer. nat. habil. Dr. h. c. Prof. h. c. Peter Scharff

Editor: Marketing Department (Phone: +49 3677 69-2520)  
Andrea Schneider (conferences@tu-ilmenau.de)

Faculty of Computer Science and Automation  
(Phone: +49 3677 69-2860)  
Univ.-Prof. Dr.-Ing. habil. Jens Haueisen

Editorial Deadline: 20. August 2010

Implementation: Ilmenau University of Technology  
Felix Böckelmann  
Philipp Schmidt

## **USB-Flash-Version.**

Publishing House: Verlag ISLE, Betriebsstätte des ISLE e.V.  
Werner-von-Siemens-Str. 16  
98693 Ilmenau

Production: CDA Datenträger Albrechts GmbH, 98529 Suhl/Albrechts

Order trough: Marketing Department (+49 3677 69-2520)  
Andrea Schneider (conferences@tu-ilmenau.de)

ISBN: 978-3-938843-53-6 (USB-Flash Version)

## **Online-Version:**

Publisher: Universitätsbibliothek Ilmenau  
[ilmedia](#)  
Postfach 10 05 65  
98684 Ilmenau

© Ilmenau University of Technology (Thür.) 2010

The content of the USB-Flash and online-documents are copyright protected by law.  
Der Inhalt des USB-Flash und die Online-Dokumente sind urheberrechtlich geschützt.

## **Home / Index:**

<http://www.db-thueringen.de/servlets/DocumentServlet?id=16739>

# IMPLEMENTATION OF A HIGH PERFORMANCE EMBEDDED VM FOR THE JAVA LANGUAGE INTEGRATING OPTIMIZATION ASPECTS OF PROCEDURAL AND FUNCTIONAL PROGRAM PARADIGMS

*Dragan Macos<sup>1</sup>, Daniel Meisen<sup>2</sup>, Sebastian von Klinski<sup>1</sup>*

<sup>1</sup>Beuth Hochschule für Technik Berlin

<sup>2</sup>CoServices Unternehmensdienstleistungen GmbH

## ABSTRACT

Software portability is one of the main goals in the world of modern mobile computing devices. Compared to other techniques such as (1) binary source-to-source compilers and (2) template based cross compiler chains, the implementation of embedded software using interpreted languages such as Java is a commonly used technique to achieve high portability of embedded software. The main challenge using the last approach however is the availability of a high performance Java virtual machine implementation for the target device(s).

Besides the functional scope reduction (i.e. CDC, CLDC profiles) the most important aspects affecting the execution performance are: (1) efficient virtual machine data structures, (2) efficient environments for the interpretation of the operational semantics and (3) an efficient optimization set.

We implemented a virtual machine for the Java programming language allowing an execution performance that is comparable with the performance of applications compiled natively using the GNU C-Compiler (GCC).

**Index Terms** – virtual machine, optimization, memo functions, indirection nodes

## 1. INTRODUCTION

There are many different virtual machines for the Java programming language that have been implemented for specific purposes in different kinds of embedded systems.

The specificity of these virtual machines is mostly determined by the definition of the functional domain, language type set and the used core library.

Our work deals with a further adjustment possibility of virtual machine implementations with regard to performance optimization in the field of embedded systems: optimization strategies for abstract machines for implementing functional programming languages.

To identify concepts that could be used in the implementation of a Java virtual machine for low profile embedded devices, to increase the runtime

performance of Java programs we analyzed techniques commonly used in procedural and functional programming paradigms.

Therefore a variety of compiler optimization techniques of procedural program paradigms such as peephole optimizations, caching, loop optimizations and of functional languages such as indirection nodes, director strings, dependency analysis, memo functions have been reviewed.

This helped to identify the most promising techniques that could be used in the implementation of a small, efficient, high-performance Java virtual machine for embedded devices. The resulting virtual machine has a very small disk footprint (50kb without the class library) that is able to execute Java programs compatible with the Java SE 1.3 specification (and the according class file format) with a performance comparable to native C programs.

We have focused on stack-based and graph-reduction-based abstract machines. The optimization methods that have been examined include memo functions and indirection nodes.

## 2. RELATED WORK

The SUN Java Virtual Machine is a stack-based virtual machine. Its specification is published by Sun [4]. The values of local variables and method parameters are stored in registers that are accessed via appropriate machine (bytecode) instructions (e.g. load and store). There are different papers published that deal with the optimization and implementation of virtual machines for the Java programming language. These optimization efforts can be subdivided into two categories:

1. Optimizing source-to-source bytecode transformations – such as peephole optimizations, static method inlining, virtual method inlining, field privatization, stack and register analysis or path profiling described in [5], etc.
2. Optimization of the virtual machine execution environment itself.

In [2] Chen and Hou define an optimized execution environment for the Java virtual machine named “Gabi”. The suggested environment includes a

couple of optimizing structures and concepts such as caching or Just-in-Time-Compiling resulting in a high-performance bytecode execution environment.

The first ideas for implementing functional languages via abstract machines were presented in Peter Landin's work [1]. Landin has defined a stack-based SECD abstract machine and its language as an intermediate form for implementing a functional language, which was a syntactic sugared version of the lambda calculus. A detailed specification of the SECD-machine is given in [7] and [8].

The main disadvantage of stack-based machines for functional languages was the “unnatural” realization of non-strict semantics via “forced” lazy evaluation: the laziness is not part of the machine semantics but must be enforced by additional commands (such as the SECD-commands *delay* and *force*). The abstract machines with “natural” non-strict semantics for interpretation of functional languages are graph-reduction-based machines (for example the SK machine and the G machine). They are further described in [6].

The SK graph reduction is discussed in the works of Paulson [10] and Wolfengagen [11].

A synthesis of functional concepts and a bytecode execution environment is presented in [3]. The model of the Java virtual machine has been built using ACL2 - a mathematical logic based on Common Lisp expressions. The goal of this work was to introduce sufficient (defensive) run-time checks to assure type-safe bytecode execution. The base optimizations for abstract machines such as the SECD stack-based machine, the SK graph reduction machine and the G-machine are further described in [6].

### 3. CONCEPTS OF ABSTRACT MACHINES

The following sections will describe the basic concepts for the realization of abstract machines.

#### 3.1. The Java Virtual Machine

The Java virtual machine is a stack-based machine. The values of local variables and method parameters are stored in registers. These registers are accessible via appropriate machine instructions.

The following method implemented in Java

```
public int add(int a, int b)
{
    int result;
    result = a + b;
    return result;
}
```

is compiled into the following bytecode-sequence

```
public add (int, int): int
    0:iload_1
    1:iload_2
    2:iadd
    3:istore_3
    4:iload_3
    5:ireturn
```

Compilers for the Java programming language transform every correct Java program into a sequence of bytecode instructions. Each instruction has a defined transition rule for the manipulation of the stack and the variable registers, describing the operational bytecode semantics:

$$\text{ins} : (S, R) \rightarrow (S', R')$$

#### 3.2. Stack-based Abstract Machines for Functional Programs

An example for a stack-based abstract machine is the abstract SECD-Machine described in the works of Peter Landin in [1], [7] and [8].

The compilation scheme for functional languages is comparable to the Java compilation scheme: a functional language is compiled into a sequence of SECD-instructions.

The abstract SECD machine consists of 4 stacks: S (stack), E (environment), C (code) and D (dump).

The operational semantics of the SECD machine is defined via a transition state system. The state of the SECD machine is defined as the states of its stacks (registers).

The operational semantic rules of the SECD machine have the following form:

$$(S, E, C, D) \rightarrow (S', E', C', D').$$

The following example shows the rule for the arithmetic addition:

$$((a \text{ b.s}), e, (\text{ADD.c}), d) \rightarrow (((b+a).s), e, c, d)$$

With '!' defining the concatenation of machine register elements and '+' resulting in the arithmetic addition.

#### 3.3. Graph Reduction Machines

An example for graph-reduction-machines is the SK-graph-reduction machine described in [6]. This machine is based on the reduction of the combinatory graph representation.

Combinators are lambda expressions without free variables. The main combinators used by the SK abstract machine are S and K combinators. Their functional definitions are:

$$Kxy \rightarrow x$$

$$Sxyz \rightarrow xz(yz)$$

The combinator K returns its first argument. The S combinator (substitution) defines functions returned by the application of the first argument to the third and applies this function to the result of the second argument applied to the third.

Every functional program can be translated into a term of SK combinators. The SK reduction machine reduces the reducible graph-representations (redex).

The not reducible SK term is the result of the functional program execution. The graphical representation of these terms is given in Figure 1.

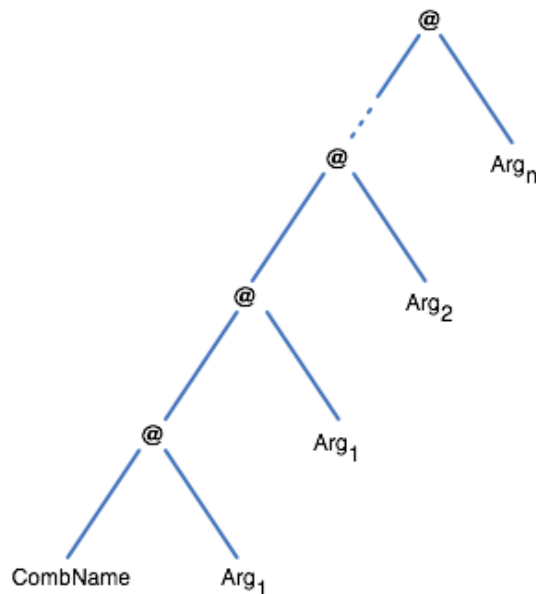


Figure 1: Combinator terms

*CombName* is the name of the combinator; *Arg<sub>1</sub>*, ..., *Arg<sub>n</sub>* are the parameters of the combinator. The reduction rule for the K combinator is shown in Figure 2.

Further references to combinators and combinatory logic can be found in [10] and [11].

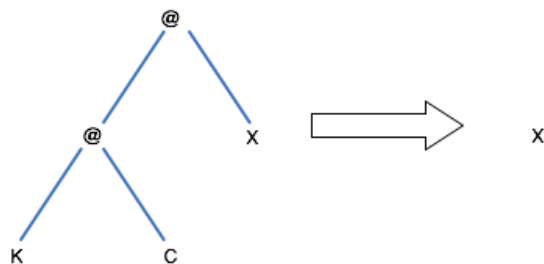


Figure 2: K combinator reduction rule

#### 4. A SELECTION OF OPTIMIZATIONS FOR FUNCTIONAL ABSTRACT MACHINES.

In the following sections a selection of optimization strategies for functional abstract machines is described.

#### 4.1. Memo Functions

The main idea of memo functions is to cache the functions and arguments they have been applied to in combination with the corresponding return values.

When the function is reapplied to one of the cached arguments the corresponding result is delivered directly without having to execute the function again. A classic example of a memoization is the Fibonacci-function [9]:

$$\begin{aligned} \text{fib } n &= 1, \text{ if } n < 2 \\ \text{fib } n &= \text{fib}(n-1) + \text{fib}(n-2), \text{ otherwise} \end{aligned}$$

The used approach to apply memo functions is to mark the functions that have to be “memoized”:

$$\begin{aligned} \text{memo fib } n &= 1, \text{ if } n < 2 \\ \text{memo fib } n &= \text{fib}(n-1) + \text{fib}(n-2), \text{ otherwise.} \end{aligned}$$

The abstract machine checks if there is a cache entry for the marked function and call parameters. If there is a previously cached result available, this result is returned without executing the function. Otherwise the function is executed and the function, call parameters and the corresponding return value are added to the cache.

#### 4.2. Indirection Nodes

Indirection nodes are used in the optimization of graph-reduction-based abstract machines for the implementation of programming languages with non-strict semantics. If a subgraph is copied, its copy doesn't contain a copy of the subgraph but only a pointer to the original subgraph, thus avoiding a second evaluation of the same graph sequence.

Figure 3 shows, how a function with lazy evaluation implies the copying of an argument (Z) that can be a redex (unevaluated expression). The copied graph is going to be executed two times.

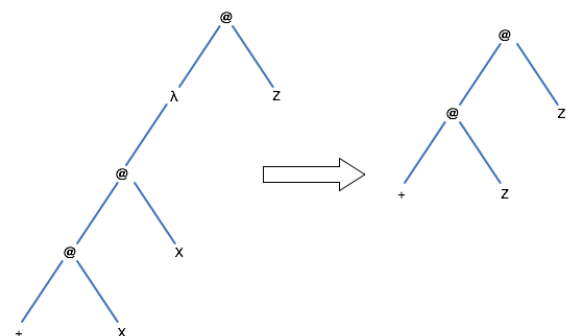


Figure 3: Lazy evaluation

The following Figure 4 shows how the duplicate execution of the graph is avoided using an indirection node (\*) pointing to the original subgraph:

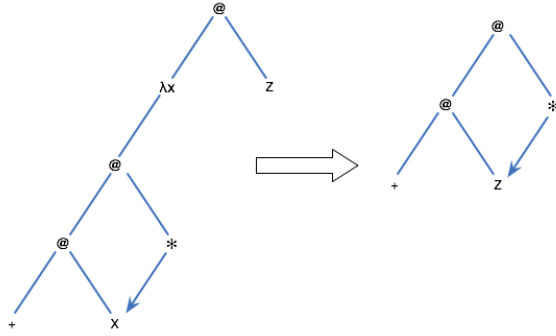


Figure 4: Indirection node

The lambda abstraction copies its first argument and the arithmetic multiplication is applied. To avoid the redundant evaluation of the copied expression, the indirection node points to the original expression.

## 5. APPLICATION OF FUNCTIONAL OPTIMIZATIONS TO A JAVA VIRTUAL MACHINE

### 5.1. Memo Functions

Applying memo functions to the field of Java virtual machines can be achieved by a simple approach: when the function activation record is called, the function name, call parameters and return type are evaluated and the cache is checked for previously memoized functions.

If the function that should be memoized needs to be marked in the application's source code, the compilation scheme would have to be altered and the memoized functions are not portable to compilers unaware of memoization. To realize high portability the following approach has been implemented:

A compiler does not have to analyze an application's source code to check if some function can be memoized. The developer decides which functions have to be memoized and adds their fully qualified names into a so-called memo function table that is part of the virtual machine runtime environment.

Additionally a list of the function's call parameters and their corresponding return values can be maintained in the memo function table within the virtual machine, allowing the caching of pre-computed return values.

As already mentioned, the main structure for the memo functions is the memo function table. In the case branch of the main interpreter loop for the execution of function calls the virtual machine interpreter implements the following approach:

- If the function is in the memo functions table:
  - the memo function call flag is set
  - the function's call parameters are taken from the argument stack and

- the lookup routine for the memo function table is called

- If the appropriate return value for the memo function is in the memo functions table, the function call is popped from the stack and the return value is put onto the virtual machine stack.
- If the memo function table does not contain the return value, the function is executed like a regular function call.
- At the end of the function execution the memo-function call flag is evaluated. If it's set, the function, call parameters and return value are added to the memo function table. The memo-function call flag is unset.

#### 5.1.1. Benchmarks

Several applications have been benchmarked with and without memo functions switched on. The execution time benefits are between 78% (Mandelbrot) and 95% (Fibonacci). The following Table 1 contains the execution times of various benchmarks that have been executed on a low profile embedded device (50 MHz ARM CPU, 8MB RAM, embedded RTOS). The benchmarks have been implemented in native C and Java using comparable language constructs. The Java virtual machine used is a clean room implementation including features like memo functions.

Table 1: Benchmark execution times

Benchmark	C	Java
	runtime (avg in ms)	
Fibonacci (memoized)	97,15	4,05
Ackermann	17,25	27,30
Mandelbrot (memoized)	33,00	3,71
Bubble Sort	6,35	4,35
Matrix	10,00	14,51
Sieve	248,85	317,25
String compare (equality)	7,70	4,52
String compare (ignore case)	7,35	8,47
String concatenation	22,50	32,01

#### 5.1.2. Disadvantages of Memo Function

The main execution time costs of the memo functions are implied by the following code sequences:

- Checking if a called function is a memo function in the virtual machine's interpreter main loop
- Checking the memo function call flag at the end of a function execution

The results of the benchmarks have shown that both code sequences do not affect the main program execution times significantly. Memo functions have been implemented as an optional optimization



measure that can be enabled/disabled using compile time switches.

### 5.1.3. Recommendation to Virtual Machine Vendors

We suggest to Java virtual machine vendors to include memo functions in the virtual machine environment because of the significant reduction of execution times. If there is only one function that can be memoized, the execution times can be rapidly minimized.

## 5.2. Indirection Nodes

A small application set has been benchmarked using indirection nodes on the SK-reduction machine. The benchmarking results showed that the SK-machine with indirection nodes is slower than the SK-machine without indirections. After examining the SK-graph structures the chains of indirection nodes could be identified as too long to significantly reduce the application execution time.

Different contexts of Java virtual machines have been analyzed to find a possible application of indirection nodes. Until now no typical situation in which indirection nodes could reduce execution time significantly has been identified. But we believe that indirection nodes can reduce the execution times of applications interpreted by a Java virtual machine because of its difference to the SK-machine. SK-machines produce many copies of SK-subgraphs that result in long indirection node chains.

Due to the “eagerness” of a Java virtual machine, it is not easy to identify typical situations for the use of indirections.

## 6. CONCLUSION

We have applied various optimization techniques of abstract machines for functional programs to a clean room implementation of a Java virtual machine. We defined an approach for the implementation of memo functions in the virtual machine and benchmarked a set of various applications. The benchmarking results showed execution time savings between 78 and 95 percent. We believe that indirection nodes can reduce execution times of the virtual machine’s main interpreter loop.

## 7. FUTURE WORK

In future projects we will investigate the possibility of applying indirection nodes to the development of Java virtual machines. Additionally the realization of delay and force instruction will be investigated to allow the implementation of lazy evaluation of selected Java virtual machine instructions.

Additionally the currently used class library implementation will be extended and further optimized.

## 8. REFERENCES

- [1] P. J. Landin, The Mechanical Evaluation of Expressions; Computer Journal 1964 6(4):308-320; 1964 by British Computer Society
- [2] Chen, F.G. Ting-Wei Hou, Design, and implementation of a Java execution environment; 1998 International Conference on Parallel and Distributed Systems
- [3] R. M. Cohen, The defensive Java Virtual Machine specification. Technical report, Electronic Data Systems Corp, 1997
- [4] Tim Lindholm, Frank Yellin, The Java™ Virtual Machine Specification; Sun Microsystems
- [5] R. Vinodh Kumar, B. Lakshmi Narayanan and R. Govindarajan, Dynamic Path Profile Aided Recompilation in a JAVA Just-In-Time Compiler; 9th International Conference on High Performance Computing – HiPC 2002, Bangalore, India.
- [6] The Implementation of Functional Programming Languages, Simon Peyton Jones, published by Prentice Hall, 1987.
- [7] Peter Henderson, Functional Programming: Application and Implementation. Prentice Hall, 1980
- [8] Olivier Danvy , A Rational Deconstruction of Landin’s SECD Machine; Lecture Notes in Computer Science; Volume 3474/2005; Springer Berlin / Heidelberg; 2005
- [9] John Hughes , Lazy memo-functions; Lecture Notes in Computer Science; Volume 201/1985; ISBN 978-3-540-15975-9;
- [10] Paulson, Lawrence C., 1995. Foundations of Functional Programming; University of Cambridge.
- [11] Wolfengagen, V.E. Combinatory logic in programming. Computations with objects through examples and exercises. -- 2-nd ed. -- M.: "Center JurInfoR" Ltd., 2003. -- x+337 c. ISBN 5-89158-101-9.
- [12] Acar, Umut A. A. et al., "Selective Memoization," Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisiana, pp. 14-25, 15-17 January 2003
- [13] Compiling Scheme to JVM bytecode: a Performance Study; Proceedings of the seventh ACM

SIGPLAN International Conference on Functional Programming; ACM Press (2002).

[14] Efficient Memoization for Dynamic Programming with Ad-Hoc Constraints Joxan Jaffar Andrew E. Santosa Răzvan Voicu; Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (2008)

[15] Partial memoization of concurrency and communication; Lukasz Ziarek, KC Sivaramakrishnan, Suresh Jagannathan; Proceedings of the 14th ACM SIGPLAN international conference on Functional programming; Association for Computing Machinery (2009).